All elemental steps that will get you started for your new life as a computer science programmer.
PADI – What is it?

PADI = Praktische Aspekte der Informatik
Welcome to the course
  ▪ How to handle C++
Includes tools like gdb, svn, etc...
Efficient use of libraries
Some bonus lessons (e.g. on documentation with doxygen etc...)

Introduction
First of all, you’ll have noticed the unusual appearance of this handbook. It’s part PowerPoint slides, part text document. I intended to use this strange mixture, because that way we will be able to use this document as a basis for our weekly classes in the CIP Pool as well as a basis for your work at home.

My experience with (power point) slides is that they are very helpful for presentations, but are of no use when you read them at home. The text passages following the slides give you detailed explanation of what these few bullets points actually meant.

In order to make things more complicated, I’ll also include the assignment sheets in this document, so you’ll have everything you need in one place. This should not prevent you from using other sources of information as well. Learning how to handle C++ is a long process, and this course is just a start. Most of the times the internet and google.com will be your helpful friend when you’re stuck somewhere.
PADI – Why?

In 1\textsuperscript{st} year, you learned JAVA

In 2\textsuperscript{nd} year, you have to pass the SEP

- You may need C++
- You need tools and skills for debugging, versioning and documentation (e.g. svn, doxygen etc).

In 3\textsuperscript{rd}, 4\textsuperscript{th}, ... year, there’s the Teamprojekt, Bachelor/Master thesis, etc.

Eventually, you will work in “the real world”

When will you acquire the skills?

**Why should I attend a course like PADI?**

...you may ask yourself.

In the past, no course like PADI existed. You learned JAVA in your first year, together with Algorithms and data structures (AuD). When people did their SEP or other courses later on, they suddenly realize they have to do some “real” programming, i.e. that they have to write a lot of code, use debuggers, profilers, etc. and may eventually have to learn a language like C++. Back in the “good old days” when students got a diploma, it was assumed that they could teach these skills themselves in their free time. That worked in most cases.

Nowadays, students have a higher workload in their curriculum. When students suddenly have to acquire these skills on their own, they fall behind schedule. So, PADI was created to assist you in your effort to become a better programmer.
How to pass this course

You should do the assignments – ideally in small groups. But(!) in the end, you won’t be graded based on your assignments.

If you already know something about C++, you may do the assignments during the 1 ½ hours we spent together twice a week, that’s just fine for me. If you’re new to C++ or programming in general, you’ll have to put some more effort into this class. But do not be afraid to do so, it is better to invest this time now instead of learning how to code when your master’s thesis is due in a month.
As a final test of your acquired skills, you will have to do a project. You will have to do it on your own (NOT in groups). By the middle of the term (probably in week 6), you will have to hand in a proposal for your final project. In this proposal, you should state, what kind of project you want to do, which techniques you used and how you are going to document it.

**PADI – How?**

**Your software project**
- Topic of your choice (more or less)
- Apply skills you learned

**You are graded based on**
- function,
- quality,
- presentation of your project
- project size (differs for Ba/Ma)
Final Remarks

Before we get started, I have got some final remarks concerning your working environment. We set up this nice CIP pool so you can work in an up-to-date work environment with (almost) everything installed you need. I realize that there are students who prefer a different working environment, e.g. their own Windows or Mac machine at home. I do not intend to “convert” anyone to a certain Operating system or Development environment. You can do the assignments on any system you like. But I do recommend you to give Linux a try, if you do not know it already. Consider the following points:

- A computer scientist should be able to work within arbitrary environments
- ...and at some point in the future you will be forced to work with a certain environment
- Your final project has to run on the CIP pool machines
- These computers are quite new and powerful
- I can only give you very limited support on other environments

What I finally recommend is that you set up your assignments in a way that they run on multiple machines, which is always a good exercise.
PADI – More Final Remarks

Be present

- Although you do not have to “earn” points in the exercises, you will definitely miss something Important
- Your attendance will be documented

You are responsible for your project!

- No way to blame others later on
Week A – C++ Basics

What is C++?

First thing first

- Compiler, Linker
- .h and .cpp files

Hello World – the usual way

Your first C++ Class

Some I/O

Assignment: Hello World – the CG way

**Week A: C++ Basics**

During this lecture there’ll be some explanation of some basic concepts of C++. It should not get too boring, so we’ll write our first program without a full understanding of what we are doing (do not worry, this happens all the time). After this first week, you’ll have your first working C++ program – wow!
What is C++?

C++ is arguably the language most often used in the IT world today. It has a lot of benefits compared to all the “new” programming languages that are around. In my opinion, learning C++ is a very important step in your IT career, since many technical concepts are concealed in other languages like JAVA. Not that this is a bad thing in general, but often it helps to understand what happens at the core of things to unleash your code’s full potential.

C++ is fast. There are only a few ways to get your code even faster than to use C++. Most of them include the use of special Hardware (GPU, FPGA and the like). It is also very powerful, you have full control over the memory used by your program. C++ also won’t tell you what to do or not to do with the code.

If there is a implementation of any algorithm on the internet, it is most likely that it’s written in C++. Many very powerful libraries are written in C++ and are readily available – often for various operating systems. Before you start programming anything in C++, it’s always worth the time to check on the net if someone already did program and publish code for your problem!

C++ is also object-oriented, you can use many powerful concepts such as (multi-)inheritance that you should have learned in your basic Programming course.

There are a lot of online and offline resources that help you if you get stuck. You should make use of them often. Here is a tiny selection of sources of information:

<table>
<thead>
<tr>
<th>What is C++ ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most superior programming language?</td>
</tr>
<tr>
<td>Fast &amp; powerful</td>
</tr>
<tr>
<td>Many libraries and much code available</td>
</tr>
<tr>
<td>Object oriented</td>
</tr>
<tr>
<td>Online help / tutorials available</td>
</tr>
<tr>
<td>Everything comes at a price</td>
</tr>
<tr>
<td>Not very intuitive / fast to program</td>
</tr>
<tr>
<td>No Garbage Collector – clean up your own mess!</td>
</tr>
<tr>
<td>Error prone</td>
</tr>
</tbody>
</table>
http://www.cplusplus.com/doc/tutorial/ - I definitely recommend this for your first steps in C++

http://www.cppreference.com/ - good reference site

http://www.learncpp.com/ - Another good introduction for learning C++

There are a lot more sites on the web, google knows them!

But there are also drawbacks when it comes to work with C++! Many people find it very unintuitive and hard to work with. There’s no built-in memory management for you, if you reserve memory for your data, you’re also responsible for freeing it afterwards. Most annoying is the fact that with many possibilities of doing things right, there come many possible ways of doing things wrong. You’ll see!
Precompiler, Compiler & Linker

Your program is (pre-)compiled and linked

- Precompile: “glue” together code
- Compile: Translate to machine-readable code
- Link: connect all compiled and external files

Different errors at different stages

- Not always easy to understand
- More on that next week
- For now, unzip weekA.zip from website, goto 00_hello directory, run make

Making your code work

Make does the compiling and linking for you

The command `make` invokes the interpretation of the Makefile in your working directory. This simple Makefile tells the compiler to compile and link the file `main.cpp` for you.

Do not worry about this too much, next week we will have some more Makefile fun.
Your first C++ program

For now, you should just go to the directory 00_hello/include in the weekA.zip file. Type in make and the program is compiled and linked.

Type ./hello to start it, you will just see a Hello World output string on the console. We will make things more interesting by writing a Hello World program the Computer Graphics way, i.e. we will create a 3D model of the globe and export it to a proper 3D file format.
Your first C++ class

Many concepts of Object Oriented programming work similar to JAVA (...and many do not). For now, let’s take a look in the directory 01_include/ to see how a simple C++ class is created. The first and major difference is that we need two files for our class: A MyClass.h and a MyClass.cpp. It is important to know that you SHOULD ALWAYS keep your declarations and definitions in separate files. Declaration means that you only define which classes and methods exist in your code, but you do not tell people what they actually do. The concept most similar to that in JAVA are interfaces. These declarations are usually stored in *.h files.

Definitions are the actual “meat” of your code, i.e., they are the implementation of your methods. For each *.cpp File, which stores the definitions of your code, a separate *.o object file is compiled by the compiler. All declarations needed by the code in a *.cpp file have to be included via the #include command. Inclusion means that the content of the actual *.h files is copied and pasted into the *.cpp file before compilation. To avoid double includes, you should always start your *.h File with a #ifndef and #define statement like in the MyClass.h. It should also always end with an #endif. This way, the precompiler (who does all the copy /paste work) only includes this file once, even if is technically included several times in the code. Keep in mind that #include statements are recursive. Also, the more you include the more time it takes to compile your code.

Your first class: MyClass.h

```cpp
#ifndef _MYCLASS_H_
#define _MYCLASS_H_

class MyClass
{
    private:
        int m_counter;
    public:
        MyClass();
        void doStuff();
};

#endif
```
You see that the function `TestClass::doStuff()` is implemented in the .cpp file. It just prints a message to the error output of your system (i.e., to your terminal window). That’s it!
And this is how you call your first C++ class, you construct it and call the `doStuff()` function ten times.
In the folder 02_structs/ you will find the example of the struct Vector3D.h. Structs are similar to Classes, except that all functions and members are public. I recommend that you only use them as data containers. More complex objects should always be modeled as classes. You may use the Vector3D as a data container to store and exchange 3-dimensional coordinates.
Your first STL data structure

In our last cost snippet in folder 03_io/ you will find everything else for your first assignment. The first thing is the usage of a vector, a data structure you will use a lot the next weeks. Vectors may store an arbitrary data structure. In the above case, it is just simple float values. You may also create a vector of Vector3D structs. The syntax would then be

```
std::vector<Vector3D> vectors;
Vector3D v;
v.x = 1.0; v.y = 2.0; v.z = 3.0;
vectors.push_back(v);
```

Of course, you would have to include the “Vector3D.h” file. Although the person who implemented the vector data structure did not know of your intention of implementing the Vector3D struct, you may use his vector to store your Vector3D structs. This is possible because of the templates mechanism used by C++.

There are a lot more STL data structures, e.g. lists, trees and dictionaries. Just have a look at any C++ reference if you need more information.
Basic I/O: MyClass.cpp readFile()

```cpp
std::ifstream fin(filename.c_str());
fin >> m_title;

int numPoints;
fin >> numPoints;

if(numPoints > 0)
{
    for (int i = 0; i < numPoints; i++)
    {
        float value;
        fin >> value;
        m_numbers.push_back(value);
    }
}
fin.close();
```

Your first file input

This is just a very basic example of reading a file. All you need to have is an `ifstream` object and its `>>` operator. With them, you can open a file and read in the data piece by piece, the `>>` operator interprets empty spaces and line breaks as separators.

You can interpret the content as any basic data structures, such as strings, integers and floats (although you should know what to expect).
**Your first file output**

*ofstreams work just the other way around as *ifstreams:* You open a file for writing (if it does not exist it is created, but the containing directory should exist). You may fill it with any type of basic data types, line breaks are added via `std::endl` (endl stands for “end line”). Note that the end line symbol can be different on different operating systems and machines.

```cpp
std::cout << "(MyClass) Writing file " << filename << std::endl;

std::ofstream fout(filename.c_str(), std::ios::out);
fout << m_title.append("*1.5") << std::endl;
fout << m_numbers.size() << std::endl;
for (int i = 0; i < m_numbers.size(); i++)
{
    fout << (m_numbers[i] * 1.5f) << std::endl;
}
fout.close();
```
Have a look at the `octaeder.ply` file in the `assignment_stub/` directory to see the syntax of the *.ply 3D files. At first a header is defined, listing the amount and properties of objects contained in the file. A list of 3D points follows, one point per row. At last, a list of lines is defined. The “3” stands for the three vertices contained (unfortunately, lines with two vertices are not supported by some blender versions) of the face. It is followed by the three indices of vertices from the point list. As a workaround, the second and third vertex are the same, effectively reducing a 3-vertex face to a 2-vertex line.

Play around with the .ply file and import it in Blender to visualize it:

**File->Import->Stanford PLY (*.ply)**
Assignment A: Hello World – the Computer Graphics Way

Task 1: Makefile [5 points]

To compile your own code, you should use a Makefile. In a later exercise, I will explain how to write a proper Makefile, for now, you can just copy-paste one of the Makefiles (e.g. from 03_io/) and modify it according to your filenames.

Task 2: World [35 points]

Write a C++ class World that has two functions:

```cpp
void readInData(const std::string filename);
void generateUVSphere(const float tessellation);
```

`readInData()` opens the text file specified by filename. This filename contains a list of coordinates (longitude and latitude), an example is given in the file `africa.txt`. It reads in the coordinates and transforms them onto a unit sphere in 3D Euclidean space.

`generateUVSphere()` generates a list of 3D coordinates for a UV sphere. `tessellation` determines how many degrees of longitude and latitude lie between each stored line.

To pass this task you should be able to print out all 3D points via `std::cout`.

*Hint:* `sin()` and `cos()` functions will help you to map the spherical coordinates to 3D space.

Task 2: Line Plotting [30 points]

Add another method to your World class:

```cpp
void plotWorld(std::string filename);
```

This functions stores a .ply file containing all stored 3D line segments. See the sample file `square.ply` for syntax details. You can view your plotted globe when you run blender and hit File->Import->Stanford PLY (*.ply).

Please note that blender is quite picky about the input text file. You will encounter import problems if the .ply file does not use UNIX line endings (might happen if you are working on a Windows machine).
Task 3: Detail [20 points]

You may have noticed that the outline of the sphere does not seem to a curved line, especially if tessellation has a high value or if a segment line of the read in data is long. During line plotting, automatically break down lines longer than a certain threshold into many smaller (curved) lines. Create a member variable of the class World that steers the level of detail during line plotting. This also apply to the coastlines of the continents.

Task 4: [10 points]

Do at least one of the following tasks to get extra points

- additional data: create data points for at least one new continent or some other interesting things on the globe (e.g.: international airline network)
- use another projection (e.g.: cylindrical)
- fill continents with triangles (this could take some time)

**Hint**

You may have to convert between radians and degree. You may have to derive the 3D coordinate from the two angles (longitude and latitude) like this:

\[
x = -\cos(\text{longitude}) \times \sin(\text{latitude}); \\
y = \cos(\text{longitude}) \times \cos(\text{latitude}); \\
z = \sin(\text{longitude}) \times 1.0;
\]
The PADI Philosophy

“Gap” knowledge
- practical skills that glue together different aspects of computer science

Heterogeneous audience
- Some of you know nothing, some know a lot
- Adjust your ambitions on the final project

Work in any environment!!

Improve your individual programming skills!

The PADI philosophy
I realize that a course like PADI comes with its individual challenges. The main goal of PADI is to teach all the “hands-on” stuff that enables you to actually write your software projects. Depending on your individual background, your PADI experience will significantly differ. Some people will struggle with the first week’s exercises and it might take more than a week for them to code them. Some of you will find most of the exercises rather easy and may even skip a few of them.

You should be honest enough to realize how far your programming skills are developed, and to adjust your ambitions accordingly. If you are a 2nd semester Bachelor student without any previous C++ knowledge, you should stick to the basics, do the exercises and propose a rather simple project. If you start with a lot of previous knowledge, you should choose a more ambitious project. This might be your chance to play with some neat technology (e.g. some powerful libraries), or to think of a bigger project. According to the ECTS scheme, you should spend 20-30 hours of study for every credit point you earn. If you don’t spend a lot of time doing the exercises, you should start with a bigger project, and you should start as soon as possible.
Project Ideas – blast from the past

PADI SoSe 2010:

- **Vegetable Vengeance**: jump and run game
- **DSA Rechner**: role-playing-game gadget
- **VTKeditor**: 3D visualization tool
- **Vokabeltrainer**: learn foreign words with google images
- **Plan B**: Zombie action game
- **QuickLauncher**: application launcher for USB sticks
- **runLog**: store and manage your running diary online
- **The Unseen Island**: 3D network game
- **XMLLayout**: store and manage rule datasets
- **Open Competition**: manage sports competitions
- **TUBS Turingmaschine**: turing machine simulator
- **Heroes of Paint and Pixel**: strategy game
- **Wortgenerator**: create words for your own language
- **KoCoReDa**: database for food/cocktail recipes
- **Spaceroids**: action game
- **PADI-Fighter**: space action game
- **Spider Attack**: tower defense game
Project Ideas – blast from the past

PADI SoSe 2010:

- **Zerobot**: IRC chatbot
- **Gicu**: Gimp and Cuda
- **Exif-Scout**: manage your photo collection
- **Loderunner**: Jump-and-run game
- **SpaceBall**: Guide a ball through a labyrinth
- **Cubenet**: share and manage data between devices
- **GeoJackal**: GeoCaching manager

PADI WiSe 2010/11

- **Hand Tracking for Applications**: Gesture Recognition
- **grade calculator**: manage school classes
- **spherefield**: 3D space shoot-em-up
- **RouteVisulizer**: Track back origins of websites
- **Direbro**: Myst-Like 3D game
- **SpaceAssault**: Side-scroller space game
- **Hike**: Pike Compiler in Haskell
- **Kartograf**: Minecraft map rendering tool
- **Kill Kenny**: Moorhuhn like game
- **GUI und Visualisierung eines DHT für DTN-Netzwerke**
## Project Ideas – blast from the past

### PADI WiSe 2010/11
- **pq-chess**: Chess game with AI
- **tetris**
- **BildBrauser**: Browser for Images
- **Mastermind**
- **Threatwork**: online board game
- **BabaMixed**: tool for planning sports events
- **FilterStudio**: drag-and-drop your filter pipeline

### PADI SoSe 2011
- **DnD offline Compendium**
- **Yet another RTS**: real-time strategy
- **nt40qt**: Chat client
- **iCal merger**: manage your calendar entries
- **SimRoute**: Visualize flight routes
- **Signal Processing Editor**
- **Eve Conquests**: Board Game Adaptation
- **Klausurenmanagementtool**
- **Scenarovision**: Image Browser and Editor
- **BadStats**: Analyze Badminton Games
- **Mango**: Web Browser
- **n-connect**: “Vier gewinnt”
Project Ideas – blast from the past

PADI WiSe 2011/12

- **Tiyan**: 3D 3rd person action RPG
- **Mineserv**: Minecraft C++ server port
- **Kamisado**: Das Brettspiel
- **Monty vs. Monsters**: Zelda-like action RPG
- **Meteroid Destroyer**: defend earth – 3D action
- **DSA-Stats**: tool for assessing pen&paper fights
- **Race-for-the-Galaxy**: card game conversion
- **Git-hydra**: visualize local git repository
- **Muppet**: control applications with kinect
- **Siedler von Catan**
- **Mining Google Insights**: analyze popularity of search terms

PADI SoSe 2012

- **NavitConfigurator**: UI design interface
- **Alchemists**: multi-player arcade game
- **SideScroller**: classic shoot-em-up
- **DevHelp**: online dev management
PADI Schedule

Hand in proposals until

27.11.2013 / 29.11.2013

Look on web page for samples
- 1-3 “pages” long
- Simple ASCII .txt files

You may start working as soon as we agree on a topic (I am always open for fresh/new ideas!)

Remember to register for the final presentation at the Prüfungsamt! (09.12.13 – 13.12.13)
Week B – Pointers, References, Inheritance

Pointers and References
Why use pointers at all?
Inheritance in C++
Assertions
Assignment: Build your own 3D house

Week B: Pointers, References and Inheritance
This week we’ll cover more advanced concepts of C++: pointers and references. You might not be familiar with these, as languages like JAVA hide these concepts from you.

We’ll also talk some more about Object Oriented programming and a useful technique called assertion.
Pointers and References

Easy to spot: & and *

References are safer to use
  - always use the `const` keyword if possible

Pointers give you more freedom, but you have to allocate and deallocate memory yourself
  - New and delete operators

Pointers and References
So far, we did not use any Pointers or references. When you code small programs that only handle small amounts of data, you do not have to worry too much about using them, anyhow.

In C++, whenever you pass an input parameter to a function, C++ implicitly copies this parameter for you, so that local changes on this variable do not affect the piece of code that called this function. With references and pointers you can actually avoid this implicit copy operation, so that functions can alter data that has been passed to them.

We’ll soon see how this is done and why we should do it in a minute.
These functions show how to use references and pointers. `float change_value_cp(float float_copy)` shows a typical function like you already know it. The parameter `float copy` is actually copied implicitly by C++, so you can make local changes to it without actually changing its original content. Note that although you may increase its value locally, the original value stays the same.

`float change_value_ref(float& float_reference)` accepts a reference as its input. You call it like you called `change_value_cp()` . The difference is that when passing a reference (instead of an implicit copy), local changes of `float_reference` also affect the original variable that has been used as the input parameter. You can avoid these changes by using const references, as in `float change_value_const_ref(const float& const_float_reference)`. The code would not compile if you try to change the value of `const_float_reference`. You may wonder why you should use a const reference at all, but it may be very desirable to avoid an implicit copy at all, i.e. if the data passed as the input argument is very large (maybe it’s a bitmap image or the contents of a video file). In this case, copying data all the time would require a lot of system resources.

The next two examples demonstrate the usage of pointers. Pointers are not data containers themselves; they merely store the actual location in your systems memory where your data is stored. E.g., a `float*` (i.e. a pointer to a float) only stores the location where a float is stored. You may access the content of a pointer (e.g., the float variable) via the `(*...)` expression, e.g. with `(*float_pointer)` in `change_value_ptr()`.

```cpp
float change_value_cp(float float_copy)
{
    return ++float_copy;
}

float change_value_ref(float& float_reference)
{
    return ++float_reference;
}

float change_value_const_ref(const float& const_float_reference)
{
    return const_float_reference+1;
}

float change_value_ptr(float* float_pointer)
{
    return ++(*float_pointer);
}

float change_value_const_ptr(const float* const_float_pointer)
{
    return (*const_float_pointer)+1;
}
```
A new concept that comes with pointers is that you have to explicitly allocate and deallocate memory for them. With the new operator, C++ is told to reserve an appropriate amount of memory for your variable and return the address to this chunk of memory for you.

In the end, you always have to delete the space used by your variable. If you delete the space and try to access its content afterwards, you may (or may not) run into weird problems, as C++ is free to use this space for other purposes. If you access the memory outside of the allocation space, the program might crash too! This is called a out-of-bound access.
A useful coding technique when working with pointers is to immediately write the invocation of delete, after declaring or allocating the pointer. This might save a lot of time later, that otherwise would be spend on looking for memory leaks.

If the pointer should be defined, but no memory should be allocated for it (e.g. because it should only be used to point to another variable) initialize the pointer to NULL, to signalize that it has no valid memory to point to.

Pointer hints

- Always write the delete call for the pointer immediately after you write the pointer declaration!
- Use the NULL pointer for initialization

```c
// Defines the pointer, but no memory is allocated yet
float* my_ptr = NULL;
// ... do something
// check if the pointer is allocated yet
if (my_ptr != NULL)
// ... the pointer memory is allocated, work with it ...
else
// ... not allocated, don’t use the pointer!

// delete works also on NULL pointers!
delete my_ptr;
```
Accessing memory locations
When a function expects a “normal” variable or a reference as a parameter, you can just pass your variable as the input parameter. When a pointer is expected, but you do not have one, you may access the memory location of your variable with the \((\ldots)\) operator. But be careful: this memory location will be deallocated automatically by C++ when leaving the scope where it was created. The current scope always ends after the next \(\{\ldots\}\) bracket is reached in the code. You may access the actual content of a pointer with the \((\ast\ldots)\) expression.

```cpp
00_pointers_references: main.cpp

float value = 0;

change_value_cp(value); // value == 0
change_value_ref(value); // value == 1
change_value_const_ref(value); // value == 1
change_value_ptr(&value); // value == 2
change_value_const_ptr(&value); // value == 2

std::cout << "memory location of value = " << ((void*)(&value));

float* value_ptr = new float(0.0);
change_value_cp(*value_ptr); // (*value_ptr) == 0
[...]
change_value_const_ptr(value_ptr); // (*value_ptr) == 2

std::cout << "memory location of value_ptr = " << ((void*)(value_ptr)) << std::endl;

delete value_ptr;
```
Why use pointers at all?

YOU can decide when your variables live and die.
You can control memory allocation
Pointers can be shared – but beware!
You can pass NULL-pointers (like in JAVA)
And there are “smart pointers”

Why use pointers?
It may seem superfluous to use pointers at all, you can do a lot with just using “regular” variables and references.

Still, it is very handy to use pointers in some scenarios. Pointers can be very useful, for modularizing your code. When a single object should be shared by different other object, they can simply share the pointer to the same memory location, you just have to make sure that none of these objects deletes the object without notifying the other ones. Additionally, when you want to control how long your variables should “live”, pointers are your weapon of choice.
HEAP and STACK?

By default, Memory is allocated on the stack.

- Memory is deallocated when you leave current scope
- You can access the memory address, but be careful with it!

If you use `new`, objects are stored on the heap

- You have to clean up with `delete`

For a thorough explanation, you may check out

Inheritance in C++

Inheritance in C++:

- Classes may inherit from multiple other classes (polymorphism)
- Classes can be abstract (like Interfaces in JAVA!)
- Eventually, you need to define your own constructors, assignment operators and destructors explicitly
- Virtual classes allow method overriding

Inheritance in C++

If you have used inheritance in JAVA, you will see that inheritance in C++ works very similar. The main difference is that you can inherit from multiple classes. This can be quite handy sometimes, but may also cause a lot of problems when things get too complicated. A good way of using inheritance wisely is to declare purely abstract classes, i.e. classes that only declare functions but do not implement them.
This is an example of a virtual, abstract class: functions are only declared, but not defined. Here, no constructor exists, but it could be possible to use one. It is also allowed to implement some of the methods, but as long as the class contains one abstract method, it cannot be instantiated.

If you put a

= 0;

expression after the declaration of a function, you tell C++ that this method is not implemented in the current class. The keyword virtual tells the C++ Compiler that the method that is executed is determined during run-time of the program and depends on the actual class type of the instance. If you use non-virtual functions, C++ decides which method to call during compile-time.

A nice overview of virtual functions in different programming languages can be found at:
http://en.wikipedia.org/wiki/Virtual_function
The class `Tree.h` does actually implement the abstract functions. Note that we need to explicitly implement a destructor, copy constructor and assignment operator because we dynamically allocate memory in the instance.

Note the virtual destructor. When a destructor of a class is invoked, it always invokes all the “down” in his line of inheritance. The keyword virtual here, makes sure that the destructor of the actual class type is invoked. Consider this example where class B is derived from A:

```cpp
B* b = new B();
A* a = b; // This works because B is also from type A

// This will only invoke the destructor of class A if the // destructor is not declared as virtual!
delete a;
```
Creating and accessing instances

Instances of classes can be created like this

```cpp
class A
{
public:
    A();
    A(const int);
    void foo();
};

A a0(5);
A a1 = A();
A* a_ptr = new A(42);
```

To access pointer instances use this

```cpp
(*a_ptr).foo();
a_ptr->foo();
```

If you access members or functions of a class, you may alternatively use the -> symbol. These two expressions are equivalent

```cpp
(*myObject).doStuff();
myObject->doStuff();
```
The Class Conifer also “overwrites” a few functions, but only three of them. The other functions are inherited from the base class Tree.

Notice here, that the destructor is not defined virtual, but it already is, because the destructor is virtual in the base class. If a class is derived from another class, all methods which are inherited and are virtual in the base class, are also virtual in the derived class, even if it is not explicitly stated.
The Rule of Three

If a class defines one of the following it should probably explicitly define all three:

- Destructor
- Copy constructor
- Assignment operator

Understand shallow copy vs. deep copy


The Rule of Three is a useful C++ programming guide line. Whenever your class needs to define a destructor, which is always the case if you dynamically allocate memory in the instance somewhere, you always need to implement a explicit copy constructor and assignment operator.

# Assertions

Make your application crash ...  
... but are very useful!

```c
#include <assert.h>

void foo(float* ptr)  
{  
    assert(ptr != NULL);  
    // Do something with ptr ...  
}
```

It is better when an application crashes in front of you than in front of a user!

**Assertions**

Whenever you code an algorithm based on certain assumptions, and you want to make sure they are fulfilled, you can declare an assumption. If you compare two vectors and assume that they are of the same length, use an assertion:

```c
assert(vector1.size() == vector2.size());
```

If you compute a probability value, you can assume that its value lies between 0 and 1:

```c
assert(probability >= 0.0 && probability <= 1.0);
```

By using assertions, you make sure that mistakes in your code get punished immediately, instead of making your program crash a few hundred lines of code later.

Assertions don’t affect the performance of your application if done right. Usually, you disable the assertions in your final release build (done via compiler flags) and so the assertion code is not included in the final binary.
Assignment_stub: Tree.cpp

```cpp
#include <cassert.h>

// Add a child object to this object
void Tree::addObject(BuildingObject* obj)
{
    std::cout << "(Tree) no objects can be inserted to a tree!" << std::endl;
    assert(false);
}
```

It may also be a good idea to give a more detailed output before you let an assertion stop your program.
Assignment B: References, Pointers & Inheritance

Task 1: Working with References [20 points]

Before you get started with your programming task, you should have a look at the contents of the folder assignment_stub inside the .zip file weekB.zip: You’ll find some new classes along with two .obj Files. .obj is another ASCII-based 3D object file. Make sure you can import these files with Blender: File -> Import -> Wavefront (.obj). You see a ground plane with some trees (not very beautiful, I admit) on it. The ground plane resembles the class Estate, the Trees resemble the classes Tree and Conifer (“Nadelbaum”). Please note that when initially compiling the code and running it, a file estate.obj is created that look like trees_nomove.obj. All the trees are located on the same location. Change that by implementing the function move (...) in Quad.cpp. This way you will learn how to handle references.

Task 2: Building a simple house – Inheritance and Pointers [40 points]

Have a look at the Estate and the Tree class to see how you create a class that inherits from BuildingObject. Implement two classes that inherit from BuildingObject: House and Storey (“Stockwerk”). An important function is getQuads(), look at this method in Tree and Estate. House should store a vector of BuildingBlocks, just like Estate. It should make sure that these objects are placed on top of each other, an instance of House does not add any Quads in the getQuads() method, it just queries all its objects (i.e. its storeys) for their Quads and places them on top of each other. A Storey is just a 6-sided box. After implementing the two classes, your .obj file should look like this.
Task 3: The roof, the roof ... more inheritance [20 points]

Have a look at the classes Conifer and Tree to see how you can further inherit from a non-virtual class. Create a class Roof that inherits from Storey and overrides the `addQuads` method. Instead of creating a box, it creates a 5-sided, pointy roof.

Make sure that you also implement the Constructor / Destructor. Also make sure that your new classes do not include each other, they should only include `BuildingObject .h`, `Quad.h` and `Vector3D.h`.

Task 4: Assertions (make you sleep well at night) [10 points]

You can make assertions that make your code more robust and less error prone. Use assertions to make sure that:

- A house only contains storeys and a single roof
- Only the topmost storey of a house is a roof
- You cannot add objects to a storey
Task 5 [10 points]

Do at least one of the following tasks to get more points

- Implement a more sophisticated type of storey that does not consist of a single cube, but may consist of several walls/columns that let you build more complex houses.
- Create bounding boxes of your BuildingObjects. Use assertions to make sure that these BoundingBoxes do not overlap (e.g., that a tree does not grow inside a house).
PADI – Ideas

Don’t have any ideas what to do with your project?

I do have!
PADI – Ideas (2)

Do something with graphics
Use Qt or OpenCV to load/manipulate/display images
  ▪ A tool for image editing (like MS Paint)
  ▪ Your own video player
  ▪ An image browser

Good ideas for beginners!
PADI – Ideas (3)

Do something using 3D graphics
Use OpenGL / 3D engines to render cool 3D stuff (Ogre3D, Irrlicht, Unity3D)
  ▫ 3D Model Viewer
  ▫ Simple Level Editor
  ▫ 3D game, e.g. racing game
  ▫ Also include physics (Bullet physics, PhysX, Havoc)

Good ideas for intermediate programmers!

See also:

http://www.opengl.org/

http://www.ogre3d.org/

http://irrlicht.sourceforge.net/

http://unity3d.com/

http://bulletphysics.org/wordpress/

http://www.nvidia.de/object/physx_new_de.html

http://www.havok.com/products/physics
PADI – Ideas (4)

Extend / Port an existing project
There are tons of projects out there
  - Look on sourceforge.net for ideas

See http://sourceforge.net/ for various open source projects.
Week C – make, cmake, gdb, IDEs

Make – What did we use the last weeks?!
CMake – Why is it better?
gdb – Advanced debugging
Eclipse – Make your life easy with IDEs
QtCreator – Another cross-platform IDE

Week C – Build environments
This week you will get to know your build environment a little better. So far, you only used the Makefiles that were provided with the code.

First, we are going to have a closer look at these Makefiles.

Then we learn how we can automatically create Makefiles with CMake.

There’s also a short introduction to the gdb (a C++ debugger).

At last, we will see how we can integrate all these things into an IDE, our weapon of choice will be Eclipse and Qt Creator.
Make – What did we use the last weeks?!

May just contain a few commands
May define different “targets”
Targets may depend on each other
May contain Macros
...not only good for C++

Use tutorials! (see below)

Make – What did we use the last weeks?!
If you need more information on Makefiles, I recommend to read one or two good tutorial, e.g.:

Chris Serson’s Makefile Tutorial


Makefiles are really great to make simple projects even more simple. They may just start very small and contain a single command. The key idea is that if you need a chain of commands to build your software, e.g. you first need to compile several files and then link them together in a last step. You can express these dependencies as targets and chain multiple targets to create new dependencies. I.e., make knows which parts of your program have to be recompiled and linked if certain source files changes.
This is a small example of a makefile (it builds the code from last week).

Let’s have a closer look

```make
CC=g++
CFLAGS=-I. -g
EXE=assignment_03

$(EXE): main.o Quad.o ObjExporter.o Estate.o Tree.o
    Conifer.o House.o Storey.o Roof.o
    $(CC) $(CFLAGS) -o $@ $^

main.o: main.cpp Tree.h Conifer.h Estate.h House.h Storey.h
    Roof.h Vector3D.h ObjExporter.h
    $(CC) $(CFLAGS) -c $<

Quad.o: Quad.cpp Quad.h Vector3D.h
    $(CC) $(CFLAGS) -c $<

[...]

Roof.o: Roof.cpp Roof.h Storey.o
    $(CC) $(CFLAGS) -c $<

clean:
    rm -f *.o *~ $(EXE)
```
MACROS
The Makefile typically starts with a list of MACROS, which are simple placeholders. Usually, you define which compiler you use, which name your project has or which compiler flags you use. That makes Makefiles reusable.

TARGETS & DEPENDENCIES
The main part of the Makefile is a list of targets, for each target a chain of commands is called. In this example, the targets assignment_03, main.o, Quad.o, Roof.o and clean are defined. Targets may depend on each other, e.g. the target assignment_03 links together compiled files that have been created by other targets (e.g., Quad.o).
There are also some shortcuts which make it easier to write short Makefiles. The line

```make
%.o: %.cpp
 $(CC) -c ^
```

Automatically compiles all object files that are needed. The drawback is that Make does not know which headers these files depend on, so if you alter a header file and run make, your changes might not be incorporated.

You can use Makefiles for other applications not related to C++ programming too, e.g. for building .pdf files from LaTeX sources.
CMake – Why it’s better

“Meta-Make”
Simple Scripting language
Works on multiple platforms with multiple build systems
  - May also create VS Solutions, Eclipse projects
Can create Installer files (.deb, .dmg, .msi)

Use tutorials! (see below)

CMake - why it's better
Here you find tutorials for CMake:

http://www.cmake.org/Wiki/CMake

It’s a good starting point for CMake basics and code snippets.

CMake is a good solution (among many others) if you want to work with bigger projects and if you would like to share your code with other people. CMake is some kind of “Meta-Make”: you use a more human-readable script-language to define your project: which files have to be compiled, which include directories are used which external libraries have to be found and linked. For many external libraries, there exist scripts that search for local installations and tell CMake where they are located on your disk.

One good thing is that you can also use CMake to create Visual Studio project files. So you can work on one project that runs on machines with Linux/GCC as well as Windows/VisualStudio.
The “basic” usage of cmake looks like this. You enter a directory that contains a CMakeLists.txt file. You type `cmake .` and Makefiles are automatically created. You may now use `make` to compile your code. The Makefile also checks if you alter the CMakeLists.txt file, so CMake is automatically invoked when you alter your project settings.

**CMake – Usage**

- **Create** `CMakeLists.txt` file
- **Run** `cmake .` in this directory (or use GUI to create Visual Studio project)
- **Or run** `cmake ..` in separate build directory
- **Run** `make` to compile
- **Delete** `CMakeCache.txt` to clean the cache (may be necessary when you move project to different location) and run `cmake .` again
- **Once Makefile is created, make also checks for updates in CMakeLists.txt**
The CMakeLists.txt file from last week looks much cleaner than the Makefile a few slides before. You state the project name, set the debugger flags, write a list of source files and tell cmake to build an executable files from all of them.
There are a lot of alternatives to CMake, if you do not like it, I encourage you to try other tools (you might discover that they are even worse). In my opinion, CMake is great because it’s not just cross-platform, it’s cross-build environments.

CMake – Alternatives

QMake
Automake/Autotools
SCons
And many others...

CMake has several advantages

- Multi-build environment (make & others)
- Multi-platform
- Deployment & Testing

http://qt-project.org/
http://www.gnu.org/software/automake/
http://www.scons.org/
Gdb – debugging for pros

A very simple but useful tutorial can be found here:

http://www.cs.cmu.edu/~gilpin/tutorial/

When your program crashes or simply misbehaves, using a debugger is a very good idea. One thing you have to make sure is that your program is compiled with the appropriate debug symbols (the option \texttt{-g} should be passed to the compiler).

All you have to is to start your program with gdb.
Call `gdb` with your binary as an argument. Type `run`.

When your assertion is violated, you can inspect your call stack and the variables that are located on every level. In this example the call stack is displayed, the user goes up three levels (3x"up") and inspects the value of `m_size->z` in the `Conifer::getQuads()` context.
In this example, the program is trapped in an infinite loop. The user presses CTRL+C to interrupt execution and displays both tessellation and phi. One can see that tessellation has a strange value, while theta is still \(-180\). From these cues one can often deduce what went wrong in the program. It is also possible to track the change of certain values over time, to set distinct breakpoints in the code, etc. But that is very tedious on the console, I recommend that you do more complicated stuff in a graphical environment.

So let's take a look at eclipse!
The following slides show how to start an Eclipse project from scratch. Before we begin, you should install all relevant plugins for eclipse. The “vanilla” eclipse only contains the needed plugins for JAVA development. You can install the C++ component using the following manual:

http://max.berger.name/howto/cdt/ar01s04.jsp#installingcdt

or (in German):

http://wiki.ubuntuusers.de/Eclipse#CDT

Please follow it step by step, it’s not that hard.

Please note that all plugin files are stored in your GITZ home directory, so make sure you have at least 26 MB of free space there. You can check with this command:

```
fs lq
```
Let’s get started with eclipse: Create a new Make C++ project.

Make sure that the “C++” mode is active (upper right part of your screen, otherwise, you might still be in “JAVA” mode).
Select the location where you unzipped the files, give a name to your project.
Create Make targets by right-clicking on the Add Make target button.
When you want to start debugging, change to the “Debug” mode.
Your screen will change to a debug setting, where you can see the call stack, the code, local variables and more stuff.

Using the buttons, you can go through the code step-by-step. You can also set breakpoints very easily. Play around with the debugger.
When you created your project with CMake, there are actually several ways to work with eclipse.

A very recent one is to create Eclipse project files with CMake. While this sounds tempting, I prefer to treat your project as a Makefile project (i.e. create Makefiles once manually and import Makefile project to eclipse).

This way you can edit your CMakeLists.txt files directly and invoke make to automatically run cmake.

---

Eclipse – fun facts

There are several ways to work with CMake and eclipse

- Create Makefiles and import C++ Make Project in eclipse
- Create Eclipse project with CMake (>2.6) http://www.cmake.org/Wiki/Eclipse_CDT4_Generator

I recommend the first one
Eclipse – Alternatives

MS Visual Studio (Windows)
QtCreator
KDevelop
Bloodshed Dev C++
Xcode (MacOS)

Why eclipse?
- Multi-platform
- Works with Make projects
QtCreator

Nice, light-weight IDE
Also supports cross-platforms (qmake)
WYSIWUG Editor for GUIs

More on that later ...
Assignment C: make, cmake, gdb, IDEs

Task 1: Make it so ... [20 points]

Go into the directory 00_make and take a look at the makefile. Run make inside the this directory to see what is happening. Do the same within the 01_make directory. Now try to change the content of a header in both directories (e.g. change line 20 in Vector3D.h from $z = val_z$; to $z = val_z*2.0$; Recompile the make target all in both folders (without running make clean). Examine the differences.

Task 2: CMake [20 points]

Go to the 02_cmake directory. Instead of a Makefile, you will find a file called CMakeLists.txt. Do at least one of the following:

(if using MacOSX / Linux) Create a Makefile with cmake. Type `cmake .` followed by the make command in this directory. The program should now be build.
(if using Windows with MS VisualStudio) Create a MS VisualStudio solution file. Use CMakes built-in GUI to configure this process.

Task 3: gdb [20 points]

Go inside the directory 03_gdb. Compile and run the code. Discover where and why the assertion is violated using the gdb.

When running again, the program does not stop. Use the gdb to find out why.

Task 4: Eclipse [30 points]

(Install and) Open Eclipse on your machine, make sure it supports C++ development. Import the already existing project 04_eclipse (select it as a C++ makefile project). Add the make target all and clean. Use the built-in-debugger to solve the problems described in task 3 (of, course, you need to use the “buggy” code from task 3 for that).

Task 5: Bonus Task [10 points]

Do at least one of the following tasks to get extra points

Use CPack to create an installer of our application for your own operating system, i.e. an rpm or deb package (Linux), a dmg package (MacOS), ar an NSIS (Windows).
Create an Eclipse project from a CMake file. Please note that you need the latest version of CMake to do that!
This week you will learn something about libraries. Libraries are nothing else but code that has been written and compiled and that can be plugged in your own programs. There are libraries for almost everything – so there is a high chance that for every problem out there, there is a library that solves it.

You can write your own libraries and distribute them. There are two options for building a library:

- **Static Libraries** (“.a” for Linux, “.lib” for Windows)
- **Shared/Dynamic Libraries** (“.so” for Linux, “.dll” for Windows)

How to use external libraries?

- OpenCV as an example

CMake and libraries
Your first library

Put .o files in an archive
  ▪ Use tool called “ar”

Distribute your .a file with your .h files
  ▪ Other people can use your code without compiling it

Hint: always provide example code
  ▪ main.cpp demonstrates basic usage

Essentially, a static library is just a bunch of compiled files put into an archive. You can pack your own library with a tool called “ar”. After that, you can distribute your .a file along with the .h header files of your project.
When taking a look at the Makefile that builds the library, you will see that the “ar” tool is called to build the lib$(LIBRARY).a target. A library file is build and linked to your main.cpp.
Manage your code

Treat each library as separate code project

- Store them in separate directories
- Use Include (I) and Link flags (L / l) flags
- Use separate Makefiles

Your main application then needs to know

- Which libraries to use?
- Where are they stored (headers and binaries)?

The previous example is somehow useless. The reason is that in many cases you won’t have all your libraries and your actual code in the same project directory. A more realistic scenario features several separate libraries in individual directories with their own Makefiles. Your actual code needs to know:

- Which libraries do you want to use?
- Where are they stored?
- Where are their headers stored?
Manage your code – 01_makestatic

CC=g++
OBJEXPORTPATH=../libobjcexport
LIBRARY=objexport
CCFLAGS=-Wall -I$(OBJEXPORTPATH) -g -c
LDFLAGS=-L$(OBJEXPORTPATH) -l$(LIBRARY) -g
EXE=assignment_04

$(EXE): main.o
   $(CC) -o $@ $^ $(LDFLAGS)

main.o: main.cpp
   $(CC) $(CCFLAGS) $<

[...]

In this example, you tell the Compiler which libraries you want to use and where they are by specifying the LDFLAGS:

-\texttt{-L$(OBJEXPORTPATH)} specifies where a library can be found
-\texttt{-l$(LIBRARY)} defines the name of the library

The path to the headers is defined in the CCFLAGS:

-\texttt{-I$(OBJEXPORTPATH)} specifies any additional directories where header files are located

You pass the CCFLAGS to the compiler and the LDFLAGS to the linker. Depending on which information is missing, you might get error messages during compilation and/or linking.
Static Libraries (.a / .lib)

Are included in your code
- Merged into each binary that uses them

Pros:
- No need for distributing additional files
- Do not change after compilation

Cons:
- Increase the file size of your binary
- When used in multiple applications
  - Code redundancy

Static libraries – pros and cons
When you use static libraries, the code parts that you use from them are merged into your binary file. This can increase the size of your binary and if you use the same library for many binaries, it can lead to redundant memory usage for your binaries. In some cases, small programs will take up several dozen MB just because they carry around large libraries. On the merit side the libraries are actually contained in your executables, so there is no need for distributing additional files (as with shared libraries) and everything that is needed to execute the binary is there. Users won’t get nasty dynamic linking errors (e.g. “dll not found”).
Shared libraries – pros and cons

The other option is building shared libraries. Shared libraries may be used by several binaries (hence the name shared). They are a good choice if you want to keep the binaries small or if a single library is reused by different applications.

It is also automatically guaranteed that if you get an update for a library, e.g. a faster implementation of it or bug fixes, your actual application will benefit from it. But it might also cease to work if the new libraries are not compatible with the old versions. When you execute your project, your system also has to know where to look for the libraries.
You can build a shared library directly with the gcc. In order to execute the binary, the library must be accessible at a predefined location. If it’s not there, you will have to specify the path to the library yourself. On Windows systems and a few Linux systems, the current working directory is also searched for libraries, but don’t rely on it. In order to add your current working directory to this path, type

```
LD_LIBRARY_PATH=./:LD_LIBRARY_PATH
```

Please note that the syntax is a little different on Macs, therefore I included several make targets in the makefile. You have to build the target MacOS all on an iMac.
Most of the time, you will use libraries written by other people. As an educative example, I chose to use OpenCV. It can be easily installed on Windows and Linux. On the machines in the CIP-pool, it should run out of the box (although it is not a bad exercise to build it on your own).

External libraries – Example OpenCV

May be installed on your system (apt-get, rpm, msi, “setup.exe”)

...or build on your own

You need:

- static or shared library
- header files
In this Makefile you will notice that several libraries are used (cv, cvaux, cxcore, highgui) that belong to OpenCV (actually, OpenCV is a collection of libraries). Depending on your system, you have to specify a different INCPATH and LIBPATH. If you build OpenCV manually, you also have to change them to the respective location (e.g. `-I~code/OpenCV/` or something like this).
With CMake it is (usually) easier to work with libraries. There is a script for every major library that searches your computer for all locations where it might be installed. If such a script finds the library, Makefiles with adjusted paths are automatically created. You should have a directory on your system where all these Find *.cmake scripts are stored. Write the path to this directory in your CMakeLists.txt file (CMAKE_MODULE_PATH). You can simply tell CMake to create static and shared libraries.

There is a .zip archive in the material for this week (folder 05_opencv_make) containing a collection of .cmake scripts.
Assignment D: Libraries

Sample code for all tasks can be found in the weekD.zip folder. In this assignment, you have to apply your new knowledge about libraries to the code from week A (The globe with Africa, remember?). There, you have to separate the main.cpp from the rest of the code (i.e., World.h and World.cpp become a library called libworld). You can find the code on our website. If you like to, you can use any other sample code.

Task 1: Make your first library [10 points]

Go into the directory 00_makelibrary and take a look at the Makefile. Run make inside his directory to see what is happening. Note that instead of linking .o object files to your main function directly, all other .o files are put in one static library called libobjexport.a. Apply the same to the code from week A.

Task 2: “External” libraries [10 points]

Go to the 01_makestatic directory. Inside, there are two subdirectories. First, go into libobjexport / and run make. Then, go into main/ and run make. Everything should be built as expected. Have a look at the Makefiles and figure out what have been changed in comparison to task A. Apply this to the code of week A.

Task 3: Shared libraries [20 points]

Go inside the directory 02_makedynamic. Compile and run the code. When executing your binary, it’ll not start properly. You have to write

```
LD_LIBRARY_PATH=.:LD_LIBRARY_PATH ./assignment_04
```

If you are working on a Mac, type

```
DYLD_LIBRARY_PATH=.:DYLD_LIBRARY_PATH ./assignment_04
```

So that local shared libraries are used. Apply the same to the code of week A. Please also note that the linking syntax is a little different on the Macs. Have a look at the Makefile for details.

Task 4: CMake and libraries [20 points]

Go into the directory 03_cmake. Note that it contains our two subdirectories and a short CMakeLists.txt. Execute cmake. and make. Everything should be build. Have a look at the CMakeLists.txt files in the subdirectories. If you understand how everything works, try your luck with the code from week A. Also try to build a SHARED instead of a STATIC library (hint: you just have to change a single word in a CMakeLists.txt).
Task 5: External Libraries! OpenCV [20 points]

Go into the directory 04_opencv. Have a look at the makefile and run make. Execute your binary and look what happens in the source code. For a basic introduction into OpenCV, go to http://opencv.org/

Task 6: CMake and OpenCV [20 EXTRA points]

Go to directory 05_opencv_cmake and have a look at the CMakeLists.txt. Unpack the CMake.zip folder somewhere and write this path to the CMakeLists.txt (CMAKE_MODULE_PATH).
This week we'll talk a little bit about Qt and its capabilities to create nice GUIs. As you might know, Qt comes with a lot of extra feature and is far more than just another GUI library. A slight drawback is that Qt needs to influence your whole build process because it introduces non C++-conform syntax in your code. Before the actual compiler starts parsing, Qt itself parses your files and automatically creates C++ conform code.
It is good to know that Qt provides everything for your programming needs: Its own build system (qmake), its own IDE (Qt Creator) and its own UI Designer Program (Qt Designer). It is even better to know that you do not have to use them if you don’t like to. You can use Cmake and your own favorite IDE/Editor if you prefer.

As there are a lot of resources to learn Qt, it may be difficult to pick one good tutorial to get started. For the exercise material, I combined several tutorials that can be useful for you, too:

Learn Qt - [http://thelins.se/learnqt/tutorial/](http://thelins.se/learnqt/tutorial/)

Qt Getting started - [http://qt-project.org/doc/qt-5.1/qtdoc/gettingstarted.html](http://qt-project.org/doc/qt-5.1/qtdoc/gettingstarted.html)

(Für Qt 4) Zetcode Qt4 tutorial - [http://zetcode.com/gui/qt4/](http://zetcode.com/gui/qt4/)
Qt – Pros and cons

Qt is ...
- Powerful
- Big
- Filled with useful extras
- Has excellent documentation

But ...
- Extends C++ standard
- You might not need everything
Qt – So what’s different?

Signals and slots

- Classes can communicate with SIGNALS and SLOTS, these can be queried/connected at runtime

Code Generators

- Qt-specific stuff is translated to standard C++
- UIC: gui-xml to C++ headers
- MOC: Qt-enhanced C++ (usually headers) to C++
- RCC: Resources into binaries

Qt introduces several new concepts to standard C++. Communication inside and between Qt-Classes is achieved with a signal and slot mechanism. For Qt 4 you can find a good introduction here [http://qt-project.org/doc/qt-4.8/signalsandslots.html](http://qt-project.org/doc/qt-4.8/signalsandslots.html)

Careful! The new Qt 5 version changed the syntax of signals and slots ([http://qt-project.org/wiki/New_Signal_Slot_Syntax](http://qt-project.org/wiki/New_Signal_Slot_Syntax)).
Counter.h

class Counter : public QObject
{
    Q_OBJECT // Important!!
private:
    int m_value;
public:
    Counter() { m_value = 0; }
    int value() const { return m_value; }
public slots: // Notice the new C++ keyword
    void setValue(int value);
signals: // Notice the new C++ keyword
    // This function is not implemented in this class!
    void valueChanged(int newValue); not implem
};

Counter.cpp

void Counter::setValue(int value)
{
    if (value != m_value) {
        m_value = value;
        emit valueChanged(value); // Emit a signal …
    }
}
In order to compile your first Qt application, you just need to include the Qt headers and link to the needed libraries. Notice that the libraries depend on the classes you use (often times you need QtCore, QtGui and QtOpenGL). Also note that the libraries might change between major Qt version (e.g. for Qt 5 you need to also link against QtWidgets).

```bash
INCPATH = -I/usr/include/qt4/QtGui -I/usr/include/qt4
LIBPATH = -L/usr/lib/

OPTIONS = -lQtCore -lQtGui
CCFLAGS = -Wall -g
EXE=qt_app

$(EXE): main.o
    g++ $(LIBPATH) $(OPTIONS) $^ -o $@

main.o: main.cpp
    g++ $(INCPATH) $(CCFLAGS) -c $<

clean:
    rm -f *.o *~ $(EXE)
```
It’s important to have a QApplication object. After you set up your application in the main method, you call QApplication::exec() to enter the Qt main loop.
If you are implementing your own Qt Class, e.g. if you derive a class from QWidget and use your own slots/signals, you have to include the Q_OBJECT keyword right after class definition. The official Qt website says:

*The Q_OBJECT macro must appear in the private section of a class definition that declares its own signals and slots or that uses other services provided by Qt’s meta-object system.*

Then you can use the extra keywords slots, signals and emit.

Before you call the C++ compiler, you have to tell the Qt Meta Object Compiler (MOC) to convert your code into standard C++ code.

```cpp
class Communicate : public QWidget
{
    Q_OBJECT

public:
    Communicate(QWidget *parent = 0);

public slots:
    void OnPlus();
    void OnMinus();

private:
    QLabel *label_number;
};
```

Please note that no “special” syntax is used in .cpp files.
When deriving a class from QObject, you have to run the MOC to get some meta information about the class. MOC parses the header file and create C++ code that lets you query information about the object, i.e., its name, the position and signature of slots. Typically, these files are named

```
moc_originalname.cpp
originalname.moc.cpp
```
or

```
originalname.moc.cxx
```

They have to be compiled and linked like any other .cpp file.

01_moc – Makefile

$(EXE): main.o moc_communicate.o communicate.o
    g++ $(LIBPATH) $(OPTIONS) $^ -o $@

main.o: main.cpp
    g++ $(INCPATH) $(CCFLAGS) -c $<

moc_communicate.o: moc_communicate.cpp
    g++ $(INCPATH) $(CCFLAGS) -c moc_communicate.cpp

moc_communicate.cpp: communicate.h
    moc $(INCPATH) $< -o @@

communicate.o: communicate.cpp
    g++ $(INCPATH) $(CCFLAGS) -c communicate.cpp

clean:
    rm -f *.o *~ $(EXE) moc_* .cpp

This is the modified Makefile, the biggest difference is the invocation of MOC to create the file moc_communicate.cpp.

Please note that both moc_communicate.o and communicate.o have to be compiled and linked.
01_moc – generated moc file

```c
static const uint qt_meta_data_Communicate[] = { //content:
    4,       // revision
    0,       // classname
    0, 0,    // classinfo
    2, 14,   // methods
    0, 0,    // properties
    0, 0,    // enums/sets
    0, 0,    // constructors
    0,       // flags
    0,       // signalCount
    // slots: signature, parameters, type, tag, flags
    13, 12, 12, 12, 0x0a,
    22, 12, 12, 12, 0x0a,
    0       // eod
};

static const char qt_meta_stringdata_Communicate[] = {
    "Communicate"
    0 "OnPlus()"
    0 "OnMinus()"
};
```

This is the content of a .moc-file. The file is changed every time you modify and moc the source header or .cpp file and therefore, it is a waste of time to edit the .moc-file by hand. As a hint for moc-files: Simply acknowledge their existence, do not forget to link them but never take any closer look at them 😊
The first step in processing a Qt application is usually to start the Qt designer and create your own user-interface layout with a few clicks. If you already have experience with GUI layouts it should be quite self-explanatory. If not, use the documentation. Your interface is stored in special Qt files, the .ui files.
You can create the contents and (very simplistic) layouts of a GUI by hand. But don’t do it! Be more productive by using an IDE!
Create your GUIs using a graphical editor, the Qt Designer. The designer is already integrated in the Qt Creator and both can be used comfortably from a common IDE.
Save as .ui (xml-based)

<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
  <class>CommunicateClassName</class>
  <widget class="QWidget" name="CommunicateClassName">
    <property name="geometry">
      <rect>
        <x>0</x>
        <y>0</y>
        <width>350</width>
        <height>190</height>
      </rect>
    </property>
    <property name="windowTitle">
      <string>Communicate</string>
    </property>
  </widget>
  <widget class="QPushButton" name="plus">
    <property name="geometry">
      <rect>
        <x>50</x>
        <y>40</y>
        <width>75</width>
        <height>30</height>
      </rect>
    </property>
  </widget>
</ui>

This is the content of a .ui-file. Similar to the .moc files, do not manually change the file, but use the Qt Designer for it.
UIC : .ui to .h

Convert your GUI into a proper C++ class

“uic communicate.ui -o ui_community.h”

Inherit from the newly created class

Use the object names of the .ui file as actual class/function names
Inherit and call setupUi

communicate.h

```cpp
#include "ui_communicate.h"
[…]
class Communicate : public QWidget, private Ui::CommunicateClassName […]
```

communicate.cpp

```cpp
[…] Ui::CommunicateClassName

Communicate::Communicate(QWidget *parent) : QWidget(parent) {
    setupUi(this); // This sets up GUI, always call it first!
    connect(plus, SIGNAL(clicked()), this, SLOT(OnPlus()));
    connect(minus, SIGNAL(clicked()), this, SLOT(OnMinus()));
} […]
```

To use a UI, you derive from the created UI class and then setup the UI using the setupUI method passing the instance in the constructor. Make sure, that you always call the method before you use the UI components.

Alternatively, you can make your class store a member of the UI-class instead of deriving from it. You still need to setup the member in the constructor.
Of course, the UIC call has to be integrated into the Makefile.

More infos on the MOC and UIC can be found (and should be read) at:

http://qt-project.org/doc/qt-4.8/moc.html

http://qt-project.org/doc/qt-4.8/uic.html
Qt – Building your application

Usual way: qmake (using Qt Creator)

Alternative - CMake
- Keywords QT4_WRAP_UI(), QT4_WRAP_CPP()

Avoid Makefiles
- Not recommended to call MOC and UIC manually
03_cmake - CMakeLists.txt

PROJECT(QtApp)
[...]

FIND_PACKAGE(Qt4 REQUIRED)
INCLUDE_DIRECTORIES(${QT_INCLUDE_DIR}
    ${QT_INCLUDE_DIR}/QtCore
    ${QT_INCLUDE_DIR}/QtGui
    ${QtApp_SOURCE_DIR})
LINK_DIRECTORIES(${QT_LINK_DIRECTORIES})
include(${QT_USE_FILE})
[...]

QT4_WRAP_UI(app_UIS_H ${app_UIS})
QT4_WRAP_CPP(app_MOC_SRCS ${app_MOC_HDRS})
INCLUDE_DIRECTORIES (${CMAKE_CURRENT_BINARY_DIR})
ADD_EXECUTABLE(qt4_app ${app_SRCS} ${app_UIS_H}
    ${app_MOC_SRCS})
TARGET_LINK_LIBRARIES(qt4_app ${QT_LIBRARIES})
Use qmake from the terminal:

1. qmake -project
2. qmake
3. make
05_advanced

Play around with application
   - Try changing SIGNAL / SLOT connections
   - Modify the *.ui file
   - Try building it with qmake

Extend the application
   - See task sheet for some ideas
Assignment E: GUIs with Qt

Task 1: Clone Brush [50 points]

First of all, build the example application (05_advanced) and make it run on your machine.

Then proceed and add two new features:

Extend the program in a way that you do have two images instead of one (so you need to have two instances of DrawingCanvas. Add the functionality that you can select a region in one of the two images (e.g., by clicking CTRL+Left Mouse Button) and then copy pixels from this spot to the other Drawing Canvas.

Two Drawing Canvas Instances. The clone brush is used to copy content into the left image.
Task 2: Color Triangle [50 points]

Take a look at the class QtColorTriangle class in the folder (06_qtcolortriangle).

You can find more details on the class interface here:


Create a project file in the Qt Creator for the files in the folder and compile the simple demo program. You should see the color triangle window.

In the next step, try to build a dynamic library that includes the QtColorTriangle class.

Afterwards add the widget to your application and use the color wheel to pick colors other than blue or red.

Also see how you can import the color wheel into your Qt Designer environment.

Week F - Optimizations

Memory leaks

- Finding leaks with valgrind

Performance optimization

- Profiling with gprof
- Profiling with callgrind

Week F - Optimizations

This week you will learn how to optimize your code. Even if your program is virtually error-free during compilation and runs smoothly when you test it, it may not perform well enough in certain cases. One reason might be that you implemented it inefficiently, so that the run-time is far longer than necessary. Another reason might be that there are so-called memory leaks, i.e. your program allocates memory without setting it free again.

You will see how you can find memory leaks under Linux with a tool called valgrind and how you can do profiling (analysis of your program’s performance) with gprof and with callgrind.

Valgrind is a collection of Linux tools and, unfortunately, you cannot use these tools for Windows. However, there are similar substitutes:

http://code.google.com/p/drmemory/

http://www.codersnotes.com/sleepy

Finding memory leaks with valgrind

Finding memory leaks is quite comfortable with valgrind. It takes a program and runs it on a simulated processor. It stores, when memory is accessed and if it’s still in use. If something seems to be wrong with your memory accesses, valgrind might help you. There are also a lot of other tools that come with valgrind, e.g. callgrind.

The drawback of valgrind is that it can be very slow. Especially when you work with large datasets, you should try to run valgrind only if you have some kind of small example program. If your application usually needs minutes to reach the critical point, it may take hours (or days?) when you evaluate it with valgrind.
You should compile your program with `-g -O0` flags:
- `-g` produces debug information
- `-O0` means no optimizations

2. Run
   ```bash
   valgrind --leak-check=yes ./assignment_06
   ```

3. Wait ...

4. Decipher leak summary

You should compile your program with `-g -O0`, but you do not have to. In fact, you can use valgrind with any software, even if you did not compile it yourself.

Once you start valgrind (see above), just wait until the program stops. All you have to do now is to decipher the leak summary.
Here you, that quite a large amount of data (almost 10 MB) is lost, further up you see that this is caused by a “cvCreateImage” in the called in the main.cpp in line 13. Someone probably forget to deallocate the memory with a cvReleaseImage().
Profiling with gprof

1. Compile with -g -pg flags
   -g Produces debug information
   -pg Generates extra code for analysis with gprof
2. Run program
   ./assignment_06
3. Program executes, gmon.out is created
4. Run
   gprof -p ./assignment_06 gmon.out
5. Interpret the “flat profile”

Profiling with gprof is a little bit different. Instead of using a virtual machine, your program runs at normal speed, every few milliseconds, the current frame stack is analyzed and the statistics are saved to a file (gmon.out). There are several ways to inspect this file:
Profiling with gprof

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>total</th>
<th>self</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37.88</td>
<td>2.11</td>
<td></td>
<td>2.11</td>
<td>100</td>
<td>21.13</td>
<td>21.13</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cgColorsteps(_IplImage const*, _IplImage*, int)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36.62</td>
<td>4.16</td>
<td>2.04</td>
<td>100</td>
<td>20.43</td>
<td>20.43</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cgColorsteps2(_IplImage const*, _IplImage*, int)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25.67</td>
<td>5.59</td>
<td>1.43</td>
<td>100</td>
<td>14.32</td>
<td>14.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cgColorsteps3(_IplImage const*, _IplImage*, int)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[...]

Hint: Note that each function is called multiple times (here 100 times) to get statistically valid timings!

When you expect the basic output of gprof (the “flat profile”), you see where most of the time is spent. Please note that gprof suffers from the fact that it only samples the code execution with a certain frequency. You can circumvent this by calling your program dozens or hundreds of times.
Profiling with callgrind

callgrind is integrated in the valgrind suite

1. Compile with -g -O0 flags
2. Run
   
   valgrind -tool=callgrind ./assignment_06

3. Program executes, callgrind.out.* is created
4. Run to generate textual overview

   callgrind_annotate callgrind.out.x

   (Alternative for graphical overview)
   
   kcachegrind callgrind.out.x

Profiling with callgrind

Another option is to use the valgrind suite for profiling. You run valgrind with the –tool=callgrind option to do this. A file callgrind.out.* is created. The * stands for the actual process ID.

Now you can run callgrind_annotate or kcachegrind to get some human-readable output. Callgrind_annotate produces a simple ASCII file, while kcachegrind gives you a graphical overview.
This is the output of kcachegrind, which displays graphically where your program spends its time. You see that cgColorsteps3() takes most of the time, but only because it saves an image to disk with cvSaveImage().
OpenCV Image Manipulation

In order to get started, here’s a very simple example how to access and manipulate image data with OpenCV. The color values of a pixel can be stored in a struct called CvScalar (which effectively stores a float array of length 4). There are get and set Methods to access/manipulate image content. Accessing image elements can be done a lot faster. Take a look at the OpenCV documentation and Q&A forum (search for “access image” or similar):

http://docs.opencv.org/modules/refman.html

http://answers.opencv.org/questions/
OpenCV Image Manipulation

```c
IplImage* load_image = cvLoadImage("./test.png");
IplImage* base_image = cvCreateImage(cvSize(load_image->width, load_image->height), IPL_DEPTH_8U, 3);

for(int y=0; y < base_image->height; y++)
{
    for(int x=0; x < base_image->width; x++)
    {
        CvScalar base_px = cvGet2D(load_image, y, x);
        CvScalar ramp_px = cvScalar(128, x%255, y%255);
        CvScalar px;
        px.val[0] = base_px.val[0]*0.5+ramp_px.val[0]*0.5;
        px.val[1] = base_px.val[1]*0.5+ramp_px.val[1]*0.5;
        px.val[2] = base_px.val[2]*0.5+ramp_px.val[2]*0.5;
        cvSet2D(base_image, y, x, px);
    }
}
```

This is just a very simple example. It iterates over all rows and columns of the image and for each pixel, it accesses the pixel value of the load_image instance, performs an operation on the pixel and writes the result to the base_image instance.
Assignment F: Optimize your Code

This week we will show you how to avoid memory leaks and slow code.

Task 1: Speedup [70 points]

Have a look at the code in assignment_stub. I wrote a function cgShadowframe that does the following:

Create a crisp shadow of a given rectangle (specified by \((x_0,y_0)\) and \((x_1,y_1)\)). The shadow of the rectangle is displaced by the amount defined in mean_kernel. Smooth that rectangle (by iterating over the neighborhood and taking the mean value, by “neighborhood” I mean all pixels whose \(x\) and \(y\) coordinate differ up to the value of mean_kernel). According to the rectangle, darken the original image. Paint a white rectangle over the shadow. The result should be a white rectangle with a shadow border at the bottom and at the right-hand side.

Your task is to modify the function cgShadowframe so that it runs significantly faster. The important thing is that the result should stay the same. You can change the way it’s created completely. You may use valgrind –callgrind (Linux only) or gprof (all systems) for this task. If you like to explore your systems toolset, you may also try any profiler you can find on your system. When using valgrind, you have to compare the total amount of assembler construction fetches to the original version of the function. On my machine, it took about \(15\ 000\ 000\ 000\) fetches to execute cgShadowframe. If gprof is your tool of choice, you have to compare against the “total ms/call” value in the flat profile. My machine spent 610 ms on the original function. Please determine these values for your own system before you start optimizing. Please always turn off automatic compiler optimizations for this comparison, you can do this by providing the compiler flag –O0.

Depending on the speedup you achieve, you’ll get more or less points

<table>
<thead>
<tr>
<th>Speedup Factor</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>10x</td>
<td>20</td>
</tr>
<tr>
<td>100x</td>
<td>40</td>
</tr>
<tr>
<td>1000x</td>
<td>50</td>
</tr>
<tr>
<td>4000x</td>
<td>60</td>
</tr>
<tr>
<td>5000x</td>
<td>70</td>
</tr>
</tbody>
</table>

On my machine, the code came down to about \(2\ 500\ 000\) instruction fetches and 0.082 ms per call.

Helpful Hints on performance optimization
In the sample code (01_profiling), I used a more efficient way of accessing image elements in an IplImage*. Instead of cvGet2D and cvSet2D you can access the pixel values directly. Avoid casting data types too often. Always think if accessing memory (in this case: pixels) is really necessary. Sometimes you do not need to iterate over the whole image for an operation. In some cases you can determine a value analytically instead of empirically (think about the blur operation).

You may use operations defined in OpenCV (and other libraries). Most of the time, these operations are implemented in a very efficient way (otherwise, these libraries wouldn’t probably be so popular).

**Task 2: Memory leaks [30 points]**

There are memory leaks in my implementation. Fix them!
Imagine you are working on a very important university project, and your deadline is tomorrow. Your program is running quite smoothly, so you decide to implement a new, very cool feature that would make your program even more awesome. Around midnight, you realize it’s not going to work as you expect. At 3 am you realize that your program does not run properly any more. At 5 am you realize your program does not run at all. At 6 am you start searching for the USB stick with the backup. At 7 am you find it, but realize the backup is more than a week old. At 8 am you have to leave for university.

Another similar situation: You work with a bunch of friends on a program. Each of you is modifying your initial prototype by implementing a cool new feature. At some point you want to merge your version, an USB stick starts making its way around the busy programmers. People start realizing that there are inconsistencies among the versions. Some changes are merged, some are not. When you continue the project next day, nobody can remember which version is the most complete one and you start again from scratch.

Some of you might have encountered these situations. The good news is that there exists a technical solution to these problems: Versioning tools. Versioning tools let you sync your data with other machines (sometimes peer-to-peer, sometimes in a client-server relationship). You can access the complete file version history, so you can, e.g., access the state of the project from last night. You can
merge your modifications with modifications made by other people, and you can get (some) assistance, when there are conflicting versions of files.

As always, there are several possibilities. The most naïve would be to simply put up an FTP server and to upload all changes to the server. There are also more mature solutions; today, we will get to know a program called subversion (or ‘svn’), that works in a strict client-server model but has several advantages over the simple FTP solution, as we will see.
SVN basics

These are examples from the freely available book Version Control with Subversion:

http://svnbook.red-bean.com/

I recommend to read Chapter 1 and Chapter 2.

You can use svn via the console or using a GUI client. For Windows, a good GUI client is Tortoise svn (http://tortoisesvn.net/). Eclipse also has a svn plugin. When working with Linux, rapidsvn is an option.

As mentioned before, the simplest approach would be to set up an FTP server. Every time a change is made to a file, the user has to upload the file to the server. This approach has several disadvantages: If the users do not check if other users made changes, they may accidently overwrite previously submitted data. In addition, no history of the files is kept.
One possible solution would be to use locks: If a user intends to work on a file, he locks it and unlocks it after the changes are submitted to the server. This prevents accidental overwrites. On the other hand, other users may have to wait for a very long time to get a lock on a file. It may also happen that one user has a lock on ‘MyClass.h’ while another user has a lock on ‘MyClass.cpp’. It is quite probable that neither of them can continue to work if the other one unlocks one of the files.
With svn, a copy-modify-merge scheme is used. Everyone is working on a local copy of the ‘repository’. Users may at any time commit changes to the repository. The repository keeps track of the files and their versions. If a user commits a file that has been changed previously by another user, the two versions are checked for conflicts. If the changes happened in different parts of the document, the files can be automatically changed. If the conflict cannot be resolved automatically, it has to be resolved by the user. The user receives the concurring versions and can resolve the conflict locally, before the file is submitted to the server.
svn basics – copy-modify-merge (2/2)

Harry compares the latest version to his own

A new merged version is created

The merged version is published

Now both users have each others’ changes
One nice benefit of svn is the unique number each revision has. If your project has reached a certain state (e.g. a specific milestone), these revisions can be tagged. You can always roll back the code to a given revision. This comes in very handy when you try to track down a bug that somehow slipped into your code. You can also display the differences between the files of your current working copy and the ones on the svn.

One word of warning: Your problems do not magically disappear when you use svn. You have to keep a disciplined workflow. That means that you should try to keep your local copy in sync with the server, that you resolve conflicts properly and do not commit files that others will not need.

svn basics – benefits

- Global revision numbers
- No serialization (parallel editing)
- Offline editing
- Merging works most of the time

But: Requires disciplined workflow and maintenance!
Initially, you check out a project with the `svn checkout` command:

```
svn checkout svn://yourname@elara.cg.cs.tu-bs.de/public/padi/
```

The current project is downloaded on your machine. You may now work on your project. E.g., you create a file called `main.cpp`. At the end of your work session you should type

```
svn add main.cpp
svn commit main.cpp
```

A text dialog may now show up where you can type in a commit message, which is quite useful. If you are working with several people on the same files, you should type

```
svn status
svn update
```

before resuming work. By doing this, you ensure that your working copy is in sync with the repository.

---

**svn workflow**

1. Update your working copy and make sure everything works properly
   - `svn update` or `svn checkout` (initial checkout)

2. Make changes.
   - `svn add`, `svn delete`, `svn copy`, `svn move`

3. Examine your changes.
   - `svn status`, `svn diff`

4. Possibly undo some changes.
   - `svn revert`
svn commands - update

$ svn update
Basic Usage
19
U foo.c
U bar.c
Updated to revision 2.

- For bug hunting it is very useful to check out old revisions with the --revision [revision-number] option!

The svn update command tells you which alterations have been (compared to your local copy). Files may have been added, deleted, updated or are in conflict. It also tells you the current revision number.
svn commands - modifying

svn add foo
svn delete foo
svn copy foo bar
svn move foo bar
svn mkdir blort
svn commands - status

$ svn status

? scratch.c   # file is not under version control
A stuff/loot/bloo.h # file is scheduled for addition
C stuff/loot/lump.c # file has conflicts from an update
D stuff/fish.c  # file is scheduled for deletion
M bar.c        # the content has local modifications

With svn status you can check the status of all local files compared to the repository on the server.
svn commands – log and diff

$ svn log
-----------------------------------------------------------
-----
line
Added include lines and corrected # of cheese slices.

$ svn diff

Index: bar.c
===========================================================
====
--- bar.c (revision 3)
+++ bar.c (working copy)
@@ -1,7 +1,12 @@
+##include <sys/types.h>
+##include <sys/stat.h>
+##include <unistd.h>
+
+##include <stdio.h>
int main(void) {
- printf("Sixty-four slices of American Cheese...
+
- printf("Sixty-five slices of American Cheese...


Every time there is a conflict, the file is modified in a way that you see the conflicting text passages. The file “sandwich.txt” has been modified in revision 2: Sauerkraut and Grilled Cheese have been added. The local copy has local modifications at the same location (Salami, Mortadella and Prosciutto). You can remove the conflict by deciding how to merge the changes and deleting the extra symbols (<<<<<<<, =======, >>>>>>>>>)

$ cat sandwich.txt

Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
<<<<<<< .mine
Salami
Mortadella
Prosciutto
=======
Sauerkraut
Grilled Chicken
>>>>>>> .r2
Creole Mustard
Bottom piece of bread
In the case of a conflict, all relevant versions of the file are stored on your local copy. The conflicting one (sandwich.txt), your locally modified one (sandwich.txt.mine), the last revision that did not conflict (sandwich.txt.r1) and the current version on the server (sandwich.txt.r2).

```
$ ls -l
sandwich.txt
sandwich.txt.mine
sandwich.txt.r1
sandwich.txt.r2
```
You have the following options:

$ svn resolve --accept theirs-full sandwich.txt

Choose this option, to replace your file with the one from the current revision completely

$ svn resolve --accept working sandwich.txt

Choose this if you corrected the conflict by hand

$ svn resolve --accept mine-full sandwich.txt

Choose this if your changes are correct

$ svn resolve --accept base sandwich.txt

Choose this to revert to the last modified revision you checked out (in his example revision 1).

It is also a good idea to clean up the temporary files *.r* and *.mine afterwards (if not automatically done by your svn application)

After you decided what to do, you can use the svn resolve command. Alternatively, you can erase all files except the original sandwich.txt and commit this one.
svn commands – commit

$ svn commit -m "Corrected number of cheese slices."
Sending sandwich.txt
Transmitting file data .
Committed revision 3.
PADI announcement

If you signed up for PADI, you do have a svn account and may check out your project:

svn checkout svn://yourname@elara.cg.cs.tu-bs.de/public/padi/yourname/

If you make changes to your proposal, edited them directly in the svn!

If you don’t get your svn username and password yet, send me a mail!
Assignment G: SVN

This week’s assignment will be mandatory! So please do it.

Task 1: Commit your project proposal [50 points]

Check out your own project’s directory:

```
svn checkout svn://elara.cg.cs.tu-bs.de/public/padi/yourname/
```

Copy your proposal (named proposal.txt) into this directory. Make sure your proposal contains your name and the Matrikelnummer.

Type

```
svn add proposal.txt
svn commit -m 'proposal added'
```

Task 2: Modify the projects.txt list [50 points]

Check out the directory

```
svn checkout svn://elara.cg.cs.tu-bs.de/public/padi/common/
```

There should be a file called projects.txt. It contains a list with all projects and current states. Please add your project information to the file, so I know who is still working on his/her project. Please update the state of your project in this file regularly. If you cancel your project for some reason, please write “cancelled” as current state.

Commit the file with your changes.
Week H - Doxygen

1. Automatic Documentation – Why?
2. Doxygen Basic Usage
3. Advanced Usage - Outlook

Week H: Documentation with Doxygen
Take a look at the “Getting Started” manual from doxygen:

http://www.stack.nl/~dimitri/doxygen/starting.html

Check out the examples for more information.
Automatic Documentation – Why?

Help others (and yourself) to understand your code better

Comment once, use for several output formats
  - Html
  - Latex
  - Custom output
  - e.g. Qt assistant

Create different views of software

No need to maintain documentation twice
Automatic Documentation – Why?

Top down vs. bottom up

**Example: MS Visio**
- Create Model first
- Create code stub automatically

**Example: Doxygen**
- Write code
- Simultaneously update documentation

What should I use?
**Doxygen Basic Usage**

Basic doxygen file easily created

- Use a GUI e.g. **doxywizard**

  doxygen -g <config-file>

Modify in text editor and run

  doxygen <config-file>
Doxygen Basic Usage

Document your code!

Several options:

- Use /*! or /** instead of /*
- Use ///! instead of //

Example:

/*! \brief Brief description.
 *     Brief description continued.
 * * Detailed description starts here.
 */
A lot of special commands:

- `\struct` to document a C-struct.
- `\union` to document a union.
- `\enum` to document an enumeration type.
- `\fn` to document a function.
- `\var` to document a variable or typedef or enum value.
- `\def` to document a #define.
- `\typedef` to document a type definition.
- `\file` to document a file.
- `\namespace` to document a namespace.
- `\package` to document a Java package.
- `\interface` to document an IDL interface.
Doxygen Example Documentation

/*! \file structcmd.h 
 * \brief A Documented file.
 * Details.
 */

/*! A test class */
class Test
{
public:
  /** An enum type.
   * The documentation block cannot be put after the enum!
   */
  enum EnumType
  {
    int EVall, /**< enum value 1 */
    int EVal2 /**< enum value 2 */
  }
  void member(); //!< a member function.
protected:
  int value; /**< an integer value */
};
Doxygen Basic Usage

Create different output

- HTML pages
- Latex files
- Graphs/Diagrams (www.graphviz.org)
Many ways to

- Include formulas (LaTeX style)
- Include graphics
- Change formatting

Example: Including images

\[ \text{\backslash image <format> <file> ["caption"]<sizeindication>=<size>} \]
Sample doxygen default project file

# Project related configuration options

# This tag specifies the encoding used for all characters in the config
# file that follow. The default is UTF-8 which is also the encoding used for all
# text before the first occurrence of this tag.

DOXYFILE_ENCODING = UTF-8

# The PROJECT_NAME tag is a single word (or a sequence of words surrounded
# by quotes) that should identify the project.

PROJECT_NAME =

# The PROJECT_NUMBER tag can be used to enter a project or revision number.
# This could be handy for archiving the generated documentation or
# if some version control system is used.

PROJECT_NUMBER =

# The OUTPUT_DIRECTORY tag is used to specify the (relative or absolute)
# base path where the generated documentation will be put.

OUTPUT_DIRECTORY =

# The OUTPUT_LANGUAGE tag is used to specify the language in which all
# documentation generated by doxygen is written. Doxygen will use this
# information to generate all constant output in the proper language.
# The default language is English, other supported languages are:
# Afrikaans, Arabic, Brazilian, Catalan, Chinese, Chinese-Traditional,
# Croatian, Czech, Danish, Dutch, Esperanto, Farsi

OUTPUT_LANGUAGE = English

[...]

# configuration options related to the LaTeX output

# If the GENERATE_LATEX tag is set to YES (the default) Doxygen will
# generate Latex output.

GENERATE_LATEX = YES

# The LATEX_OUTPUT tag is used to specify where the LaTeX docs will be put.
# If a relative path is entered the value of OUTPUT_DIRECTORY will be
# put in front of it. If left blank 'latex' will be used as the default
# path.

LATEX_OUTPUT = latex
SVN Account Reminder

Everybody should have a SVN account by now!

If you do not have a SVN account, send me a mail!
Week I - Final Preparations

Dates for the final presentation:

1. Block
   **Friday, 24. January 2014 / 12:45 – 15:00**
2. Block
   **Friday, 31. January 2014 / 12:45 – 15:00**

You need to be there for the whole block you are assigned to.
Your presentation

Presentation:

- 2 projectors ("beamer"):  
  - One for presentation slides  
  - One for your live demo  
- Include additional slides into your presentation  
  - 3-4 slides (not more)  
- Place slides in common folder on the SVN  
  - Until two days before your presentation
Your presentation

Structure your presentation

- Who are you?
- What is your project about?
- What are the core features (show them)?
- Why did you choose the project?
- What kind of difficulties did you encounter?
Your presentation

Remarks:

- Resolution:
  - Slides 1024x768 px
  - Demo 1280x1024 px
- Your program must be executable from any account of the CIP-Pool (!)
- Put ALL necessary files into the svn
Your presentation

”No-no”s and “Do-not”s:

- Imprecise, unstructured talks
- Quick and dirty look
- Keep answering questions with “I don’t know”
- Excess 10 minutes of everybody’s time
  - Plan your talk to be 5 minutes
Grading

Your grade is influenced by:

- Your program runs
- Your program runs smoothly
- You show all mandatory features described in your proposal (do not expect that we ask for them if you don’t show them)
- Your talk and your software “look good”
Dress Rehearsal

Meet-up on Wednesday (22.01.2014 for Block A and 29.01.2014 for Block B) before each colloquium block

- Check if your software runs
- See if there are technical problems
- Eventually, talk about your slides